Multi-Issue Butterfly Architecture for Sparse Convex Quadratic Programming

Maolin Wang
The Hong Kong University of Science and Technology
maolinwang@ust.hk

Ian McInerney
Imperial College London
i.mcinerney17@imperial.ac.uk

Bartolomeo Stellato
Princeton University
bstellato@princeton.edu

Fengbin Tu
The Hong Kong University of Science and Technology
fengbintu@ust.hk

Stephen Boyd
Stanford University
boyd@stanford.edu

Hayden Kwok-Hay So
University of Hong Kong
hso@eee.hku.hk

Kwang-Ting Cheng
The Hong Kong University of Science and Technology
timcheng@ust.hk

***Abstract*—Convex quadratic optimization solvers are extensively utilized in various domains; however, achieving optimal performance in diverse situations remains a significant challenge due to the sparse nature of objective and constraint matrices. General-purpose architectures struggle with hardware utilization when performing critical sparse matrix operations, such as factorization and multiplication. To address this issue, we introduce a pipelined spatial architecture, Multi-Issue Butterfly (MIB), which supports all primitive scalar, vector, and matrix operations required by the Alternating Direction Method of Multipliers (ADMM) based solver algorithm.**

The proposed architecture features a butterfly computational network with innovative working modes for each node, controlled by runtime instructions. We developed a companion scheduling method for matrix operations based on their sparsity patterns. For factorization, an elimination tree guides the network instructions reordering to avoid data hazards caused by computation dependencies. For matrix-vector multiplication, data prefetching resolves structural hazards caused by read and write conflicts to register files. Instructions without hazards are issued simultaneously to increase pipeline throughput and function unit utilization.

We evaluate the proposed architecture using FPGA prototypes, representing the first fully FPGA-based generic QP solver. Our assessment includes extensive performance and efficiency benchmarks across 100 QP problems from five application domains. Compared to the same algorithm variation running on CPU backends, our prototype achieves a geometric mean of $30.5\times$ end-to-end speedup, $127.0\times$ greater energy efficiency, and $16.5\times$ less runtime jitter. In comparison to GPU backends, the prototype attains a geometric mean of $4.3\times$ faster end-to-end speedup, $21.7\times$ higher energy efficiency, and $33.4\times$ less runtime jitter.

***Index Terms*—convex optimization, quadratic programming, out-of-order instruction issue, FPGA**

I. Introduction

Convex quadratic optimization or Quadratic Programming (QP) is an essential mathematical optimization technique with immense significance across numerous fields. Various applications of QP can be found in diverse domains, such as extracting neural activities from in vivo calcium imaging [16], modeling sensor fusion problems in visual-inertial odometry [10], performing backtesting in portfolio optimization, which involves solving millions of quadratic programming problems in a day [8], and as the subproblem inside the popular Sequential Quadratic Programming method for solving general nonlinear optimization problems [7]. QPs can also be embedded in deep neural networks, replacing convolutional or fully connected layers to model more complex data dependencies [1], [3]. Quadratic programming with n decision variables and m constraints can be formulated as

$$\begin{aligned} &\text{minimize} && (1/2)x^T P x + q^T x && \text{(1a)} \\ &\text{subject to} && l \le Ax \le u, && \text{(1b)} \end{aligned}$$

where $x \in \mathbb{R}^n$ is the vector of decision variables, the positive semi-definite matrix $P \in \mathbb{S}^n_+$ and vector $q \in \mathbb{R}^n$ define the objective, and the matrix $A \in \mathbb{R}^{m \times n}$ and vectors $l, u \in \mathbb{R}^m$ describe the constraints.

The success of applications relying on QPs heavily depends on the speed and energy efficiency of solvers. For instance, applying Model Predictive Control to systems with millisecond-scale sampling periods, such as power converters [33] or turbomachinery [43], requires solving a QP after each sensor sample to compute the next control command and require a deterministic solver runtime to ensure stability by guaranteeing the control command is applied before the next sensor sample, while other applications like financial backtesting [14] require solving sets of hundreds of QPs with varying parameters every hour/day to optimize investment portfolios.

After transforming [15] the original problems from various domains into the standard QP formulation shown in (1), the inherent structures of specific application domains are preserved as sparsity patterns of the objective matrix P and constraint matrix A, which are defined by the locations of non-zero values within them. The sparse nature of the objective and constraint matrices poses a significant challenge in efficiently accelerating the solver algorithm. The requirements for effectively accelerating the core operations in the solver algorithm,

This research was supported by ACCESS – AI Chip Center for Emerging Smart Systems, sponsored by InnoHK funding, Hong Kong SAR.

primarily sparse matrix *factorization* and *multiplication*, will vary for each problem.

Solvers running on general-purpose platforms like CPUs [38] and GPUs [35] experience low hardware utilization due to the high sparsity ratio (usually over 99% zeros) and irregular sparsity patterns of problem matrices. To achieve strict performance goals in terms of speed, energy efficiency, and reliability, applications can utilize custom hardware solvers [9], [20], [22], [23], [26] that take advantage of prior knowledge of matrix structures for specific problems. While this approach delivers the necessary performance, designing customized solvers demands significant hardware expertise, and the resulting hardware solver may not be easily transferable between application domains.

In this work, we propose a pipelined spatial architecture that enables sparse pattern-specific acceleration of core matrix operations without sacrificing flexibility. The proposed architecture features a computational network design in which each node can be controlled through instructions for either computation or routing. The design employs a butterfly topology, which allows for the integration of various primitive sparse operations within a single clock cycle, thus improving spatial utilization. Additionally, we develop companion instruction scheduling addressing structural and data pipeline hazards and improve temporal utilization. By *temporally and spatially interleaving* network instructions, our architecture achieves compile-time sparsity specificity for matrix multiplication and factorization with various sparsity patterns, making it suitable for solving QP workloads with diverse sparse structures.

We present an FPGA-based prototype system using this architecture, and a complete compiler stack for mapping applications to the best possible variation of the solver algorithm and performing out-of-order instruction scheduling based on the sparsity structure of the problem. This prototype is the first fully FPGA-based generic QP solver.

We conducted extensive performance and efficiency benchmarks across a wide range of applications to show the generalizability and efficiency of the proposed architecture. Compared to the same algorithm variation running on CPU backends, our prototype achieves a geometric mean of $30.5\times$ end-to-end speedup, $127.0\times$ greater energy efficiency, and $16.5\times$ less runtime jitter. When compared to GPU backends, the prototype attains a geometric mean of $4.3\times$ faster end-to-end speedup, $21.7\times$ higher energy efficiency, and $33.4\times$ less runtime jitter.

The remainder of this paper is organized as follows: Section II introduces two solver algorithm variations and their computational characteristics. Section III presents our architecture design that supports the core operation set required for different algorithm variations. Section IV describes the multiple-instruction issuing for scheduling sparse operations on the proposed architecture. Section V evaluates our FPGA-based prototype across a wide range of applications. Section VI discusses related works on the acceleration of relevant solver algorithms. We conclude our work in Section VII.

Algorithm 1 OSQP algorithm

```
1: given:
       initial values $x^0$, $z^0$, $y^0$ and parameters
       $\rho > 0$, $\sigma > 0$, $\alpha \in (0, 2)$
2: repeat
3:     solve
           $\begin{bmatrix} P+\sigma I & A^T \\ A & -\rho^{-1}I \end{bmatrix} \begin{bmatrix} \tilde{x}^{k+1} \\ \nu^{k+1} \end{bmatrix} = \begin{bmatrix} \sigma x^k - q \\ z^k - \rho^{-1}y^k \end{bmatrix}$   (2)
4:     $\tilde{z}^{k+1} \leftarrow z^k + \rho^{-1}(\nu^{k+1} - y^k)$
5:     $x^{k+1} \leftarrow \alpha\tilde{x}^{k+1} + (1-\alpha)x^k$
6:     $z^{k+1} \leftarrow \Pi(\alpha\tilde{z}^{k+1} + (1-\alpha)z^k + \rho^{-1}y^k)$
7:     $y^{k+1} \leftarrow y^k + \rho(\alpha\tilde{z}^{k+1} + (1-\alpha)z^k - z^{k+1})$
8: until the termination criterion is satisfied
```

II. The QP Solver Algorithm

Our work employs the method used in the Operator Splitting Quadratic Program (OSQP) solver [38], which is based on the alternating direction method of multipliers (ADMM) and is well-suited for both embedded and large-scale optimization problems. In this section, we introduce two variations of the algorithm with distinct computational characteristics suitable for different application scenarios. Additionally, we outline the core operation set and break it down to four primitive computation patterns that govern the runtime of these two variations.

- Multiplication and Accumulation (MAC)
- Permute vector elements across register files
- Column elimination
- Element-wise multiplication, addition and subtraction, etc

A. Overview

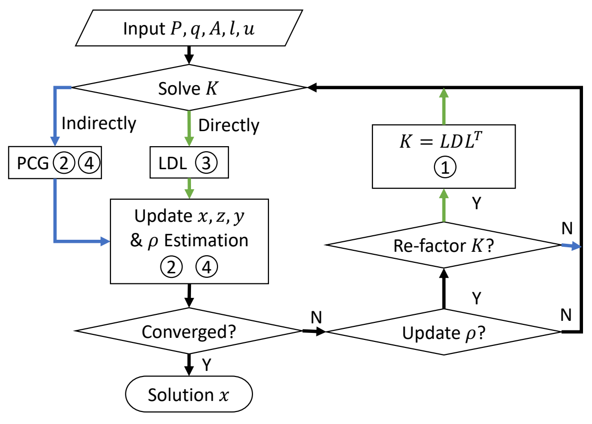
To solve the QP (1), OSQP introduces a new decision variable $z \in \mathbb{R}^m$ to convert the inequality constraint (1b) into the pair of equality/inequality constraints $Ax = z$ and $l \leq z \leq u$, and then introduces two auxiliary variables, $\tilde{x} \in \mathbb{R}^n$ and $\tilde{z} \in \mathbb{R}^m$. The steps of the OSQP algorithm then consist of alternating between solving two subproblems: (i) solving an equality-constrained QP by solving the linear system (2), and (ii) the Euclidean projection $\Pi(\cdot)$ of z onto the new inequality constraint set $z \in [l, u]$.

The first subproblem is formed using the Lagrange multiplier method, with $y, v \in \mathbb{R}^m$ the Lagrange multipliers of the constraints, to form the equality-constrained QP

$$\begin{aligned} \text{minimize} \quad & (1/2)\tilde{x}^T P \tilde{x} + q^T \tilde{x} + (\sigma/2)\|\tilde{x} - x^k\|_2^2 \\ & + (\rho/2)\|\tilde{z} - z^k + \rho^{-1}y^k\|_2^2 \\ \text{subject to} \quad & A\tilde{x} = \tilde{z}. \end{aligned}$$

The optimal solution to this subproblem can be found by solving the linear system (2), known as the KKT linear system containing the KKT matrix K, defined as

$$K := \begin{bmatrix} P+\sigma I & A^T \\ A & -\rho^{-1}I \end{bmatrix}. \qquad (3)$$



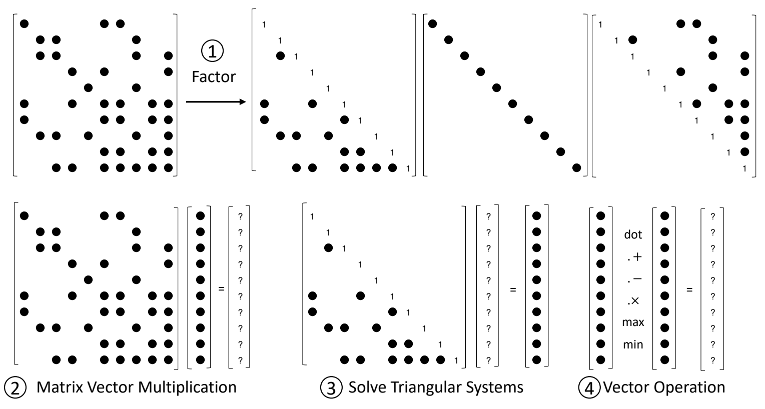


Fig. 1. The left part shows the computation flow of the solver algorithm. The right part shows the bottleneck operation set.

The introduction of the step size parameters $\rho > 0$ and $\sigma > 0$ by OSQP ensures that K always has full rank, and therefore (2) always has a solution. The values of σ and ρ impact the convergence speed of the solver, and their optimal values may vary based on the problem, therefore OSQP periodically adjusts the step size ρ while running to ensure a fast convergence of the overall algorithm.

The OSQP algorithm and its simplified computation flow can be seen in Algorithm 1 and Figure 1, respectively, with more details in [38]. It is important to note that solving the KKT linear system (2) is the most computationally demanding step, and that the sparse structure of K presents a significant challenge for efficient hardware acceleration of this step.

B. Application-specific Sparsity Patterns

The sparsity of the problem matrices arises from the inherent structure of specific application domains. The sparsity pattern of the matrix remains constant across various problem instances, despite differences in numeric values. Take for example portfolio optimization, where the objective is to optimize the risk-adjusted return of the portfolio. The weight of each asset is represented by $x \in R^n$, while the expected return of each asset is denoted by μ. The risk aversion parameter is represented by γ, and the correlation among different assets, along with the asset-specific risk, is indicated by $D \in R^{n \times n}$. The factor load matrix is represented by F, and factors are denoted by y. The base form of the portfolio optimization problem is then

$$\text{minimize} \quad x^T D x + y^T y - \gamma^{-1} \mu^T x \tag{4a}$$

$$\text{subject to} \quad 1^T x = 1, \tag{4b}$$

$$y = F^T x, \tag{4c}$$

$$x \geq 0, \tag{4d}$$

where the first constraint ensures the normalization of the portfolio (the sum of all asset weights equals 1), the second employs a factor model, denoted by F^T, to represent the correlation among assets, and the third prohibits short selling.

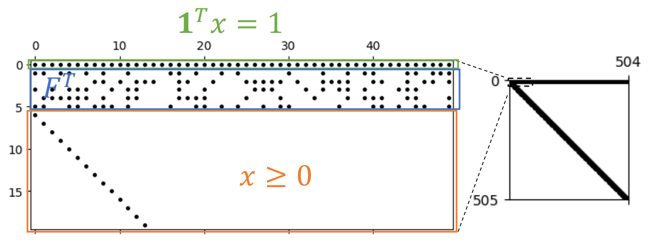


Fig. 2. Sample sparsity pattern from the portfolio optimization domain. This pattern is constant across different problem instances.

After transforming the problem into the standard QP formulation shown in (1), the three constraints are preserved as distinct blocks in the matrix A, which is further embedded into K. The sparsity of this combined block structure is shown in Figure 2, where the overall A matrix is a half-arrow format with a block of rows at the top and values along a diagonal. The determination of the optimal portfolio requires solving millions of QPs, each with a distinct γ. Note that the sparsity patterns of matrices are constant across all problem instances.

The structure and sparsity pattern of the A matrix for various domains are shown in the top row of Figure 3. Based on the sparsity ratio and the sparsity pattern of K in the different domains, we can choose to solve the KKT linear system either *directly* or *indirectly*, resulting in two variations of OSQP.

C. OSQP-Direct: Factorization-based KKT Solver

The direct method of solving (2) factors K into the form LDL^T, with L lower triangular and D diagonal, using the LDL decomposition. This factorization can be done recursively using an up-looking factorization that grows L row by row. Assuming we know the factorization of the $(n-1) \times (n-1)$ submatrix $K_{11} = L_{11} L_{11}^T$, the original factorization can be rewritten in a block matrix format as shown in (5). The factors L can then be found by solving for l_{22}, d_{22} and

l_{12}, where $l_{22} = \sqrt{k_{22} - l_{12}^T l_{12}}$, l_{12} comes from solving the triangular system $L_{11} l_{12} = k_{12}$ for l_{12}, and extracting d_{22}. The boundary case of the recursion $n = 2$ can be easily solved as L_{11} is a scalar.

$$\begin{bmatrix} L_{11} & 0 \\ l_{12}^T & l_{22} \end{bmatrix} \begin{bmatrix} D_{11} & 0 \\ 0 & d_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & l_{12} \\ 0 & l_{22} \end{bmatrix} = \begin{bmatrix} K_{11} & k_{12} \\ k_{12}^T & k_{22} \end{bmatrix} \tag{5}$$

In the sparse case, The LDL factorization is split into two parts: (i) the symbolic factorization that uses the sparsity pattern of K to compute the location of the non-zero entries in L, and (ii) the numeric factorization that actually computes all the non-zero entries in L and D. Once the numeric factorization is computed, the solution to the original linear system can be easily found through forward-backward substitution using two triangular system solvers. Since ρ is a component of the KKT matrix K, whenever ρ is updated during the ADMM iterations, K needs to be numerically refactored again (but not symbolically refactored).

The dominant operation within this factorization is solving triangular linear systems, where the dimension of the system grows by one in each iteration. There are two ways of solving the triangular linear systems: row-based and column-based.

Consider solving the $n \times n$ example in (6) in a row-based manner. The solution vector x can be computed sequentially using substitution, and so to compute x_1, we only need to consider the first row of L, which contains only one non-zero element. Once x_1 is determined, we can substitute its value into the equation related to the second row of L to compute x_2.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ l_{21} & 1 & 0 & 0 & \cdots & 0 \\ l_{31} & l_{32} & 1 & 0 & \cdots & 0 \\ l_{41} & l_{42} & l_{43} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ l_{n1} & l_{n2} & l_{n3} & l_{n4} & \cdots & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \\ \vdots \\ b_n \end{bmatrix} \tag{6}$$

This process can be continued for the remaining rows of L until all the elements of x are computed. Step i of row substitution requires access to the ith element of b and the i previously computed elements of x. Since we utilize the LDL factorization, we can assume all the diagonal elements of L are 1, and then express the ith step as

$$x_i = b_i - \sum_{j=1}^{i-1} l_{ij} x_j. \tag{7}$$

Multiplication and accumulation are therefore the major operations involved in the row-based method.

Alternately, the substitution can also be done in a column-based manner. In this case, (7) can be written as the set of equations (8) to (12). Instead of computing these equations in sequence, we can also group computation in each equation based on x_i and then perform column substitution. The column substitution for step i requires access to the last i elements of b and the ith element of x. The parallel access to vector b is crucial, and the problem can be modeled similarly to the access to vector x in the row substitution case. *Column elimination* in the following equations is the major operation involved in the column-based method.

$$x_1 = b_1 \tag{8}$$

$$x_2 = b_2 - l_{21} x_1 \tag{9}$$

$$x_3 = b_3 - l_{31} x_1 - l_{32} x_2 \tag{10}$$

$$x_4 = b_4 - l_{41} x_1 - l_{42} x_2 - l_{43} x_3 \tag{11}$$

$$\cdots \tag{12}$$

Algorithm 2 Preconditioned Conjugate Gradient

```
1: given:
       initial values $x^0$ and preconditioner $M$
       $S := P + \sigma I + \rho A^T A$
       $b := \sigma x^k - q + A^T(\rho z^k - y^k)$
       $r^0 := S x^0 - b, \quad d^0 := M^{-1} r^0, \quad p^0 := -d^0$
2: repeat
3:     $\lambda^{i+1} \leftarrow \frac{(r^i)^T d^i}{(p^i)^T S p^i}$
4:     $x^{i+1} \leftarrow x^i + \lambda^i p^i$
5:     $r^{i+1} \leftarrow r^i + \lambda^i S p^i$
6:     $d^{i+1} \leftarrow M^{-1} r^{i+1}$
7:     $\mu^{i+1} \leftarrow \frac{(r^{i+1})^T d^{i+1}}{(r^i)^T d^i}$
8:     $p^{i+1} \leftarrow -d^{i+1} + \mu^{i+1} p^i$
9:     $i \leftarrow i + 1$
10: until $||r^i|| < \epsilon ||b||$
```

D. OSQP-Indirect: PCG KKT Solver

Alternatively, we can solve the KKT system (2) using an iterative method, such as the Preconditioned Conjugate Gradient (PCG), giving the OSQP-indirect solver variant. The quasi-definite KKT matrix (3) can be reduced to a positive definite matrix through block elimination, forming

$$S := (P + \sigma I + \rho A^T A),$$

which then forms a positive definite linear system that can be solved using the PCG shown in Algorithm 2.

An iteration of PCG requires a matrix-vector product with S, and other vector operations such as element-wise multiplication and dot product computation. When using the PCG to solve the KKT system (2), S should never be explicitly computed, since forming the matrix-matrix product $A^T A$ could destroy any sparsity that the problem originally possessed and require storing a dense matrix for S. Therefore, it is better to store P, A, A^T separately and then incrementally perform the matrix-vector multiplication to compute the matrix-vector product with S.

E. Core Matrix Operation Set

The core matrix and vector operations involved in the two OSQP variants (with either the direct or indirect KKT solver) are highlighted in the right part of Figure 1. To examine the differences in total computation cost of these two variants while solving optimization problems arising from different domains, we analyze the computational characteristics of the benchmark problems from [38], which encompass 100 real-world QP problems from five application domains: portfolio optimization, Lasso, Huber fitting, model predictive control (MPC), and support vector machines (SVM). Each domain includes 20 problems of varying scales, characterized by the total number of non-zeros in matrices A and P. The total floating-point operations (FLOPs) for the two algorithm variants, as well as the breakdown of the core operations broken down into the four primitive operations: multiplication and accumulation (MAC), vector permutation across register files, column elimination, and element-wise multiplication, are shown in Figure 3.

From this, we can see that the different applications have different operation profiles. For instance, for the OSQP-direct solver variant, the portfolio optimization problem spends more FLOPs on triangular system solves than the factorization, while Huber fitting spends almost all its FLOPs on the factorization and very little in the triangular solve. We can also see that the variant requiring more FLOPs also depends on the application, with LASSO and MPC using nearly the same amount of FLOPs for both indirect and direct for some problems, while portfolio optimization uses more for the indirect and Huber fitting uses more for the direct variant.

Our goal in this work is to design a compute architecture that allows the solver to efficiently support different application sparsity structures without requiring static hardware reconfiguration. In the next section, we introduce our novel architecture design, which achieves sparsity pattern-specific customization through *temporal and spatial* combinations of these primitive operations.

III. Architecture and System Design

The high degree of sparsity of the problem matrices renders the core operation set highly memory-intensive. Modern memory technologies, like High Bandwidth Memory (HBM), provide substantial bandwidth capacity and reduced power consumption through 3D stacking. However, to achieve this high bandwidth, contiguous data access patterns are necessary. The irregular data flow of sparse matrix operations in solver algorithms poses considerable challenges for architectural design.

Consider the case of multiplication within the core operator set, where the matrices and vectors to be multiplied are stored in the HBM. The matrix is usually stored in a compressed format, such as Compressed Sparse Column (CSC), which allows for contiguous access to non-zero values. However, vector access becomes random in this situation, leading to a slowdown in overall throughput. The primary objective of high-performance architectural design is to ensure both access patterns are contiguous and fully utilize the available memory bandwidth.

A. Baseline architecture for the MAC primitive

We begin with a baseline architecture supporting the MAC primitive operation, as depicted in Figure 4. The vector elements are distributed to various register files in advance via a network, enabling random access capability. A multi-mode MAC tree is then utilized to accumulate partial sums of the multiplication of one matrix row with the vector, or whole sums of multiple rows that are less than the width of the MAC tree. Another routing network aligns the varying number of accumulation results produced during each clock cycle.

Letting the maximum number of data items that can be obtained from the HBM in every clock cycle be C under the contiguous access pattern, we use this as the unified scalability parameter to control the width of all architecture components. Different widths can be instantiated to achieve a trade-off between resource usage and performance.

The yellow connections within the MAC tree can be added or removed at hardware design time to provide static customization for multiplication operations for matrices with specific sparsity patterns. Instantiating all the yellow connections leads to the best MAC throughput per clock cycle for all possible matrix sparse patterns. However, this increases routing difficulties and may lead to a drop in the max frequency.

The input and output alignment networks execute high-throughput C-to-C permutations to preserve the contiguous access pattern to the HBM. A widely-used butterfly structure is adopted for the input and output network topology, as it can efficiently manage various permutation workloads with minimal connection complexity. With only two input and output connections for each node in every stage, this topology offers easy scalability.

B. Computational Network Design for All Primitives

We enhance the baseline design to accommodate all primitive operations, not just MAC, by leveraging a key observation: the butterfly topology inherently supports the multi-mode MAC tree structure.

This insight inspired our computational network design, which supports all the primitives required by the core operation set. As depicted in Figure 5a, each node in our proposed network can operate in four different modes during runtime. This design allows us to integrate the MAC tree within the butterfly network and consolidate the three architecture components shown in the right part of Figure 4.

As illustrated in Figure 5b, the resulting architecture design enables fine-grained, instruction-based control during runtime. Each adder node can function in one of four modes, while input and output multiplier nodes can be bypassed if needed. By issuing network instructions alongside data items from the HBM, the network can be adapted to support all primitive operations.

C. Network Instruction for Primitive Instructions

In every cycle, C items are selected from random locations within each register files (`R0, R1, ..., RC`) and enter the network. The addresses of the items are specified by the network instructions. These items undergo $\log C$ processing stages before being written back to random locations in their respective banks. Utilizing a fully pipelined design, the

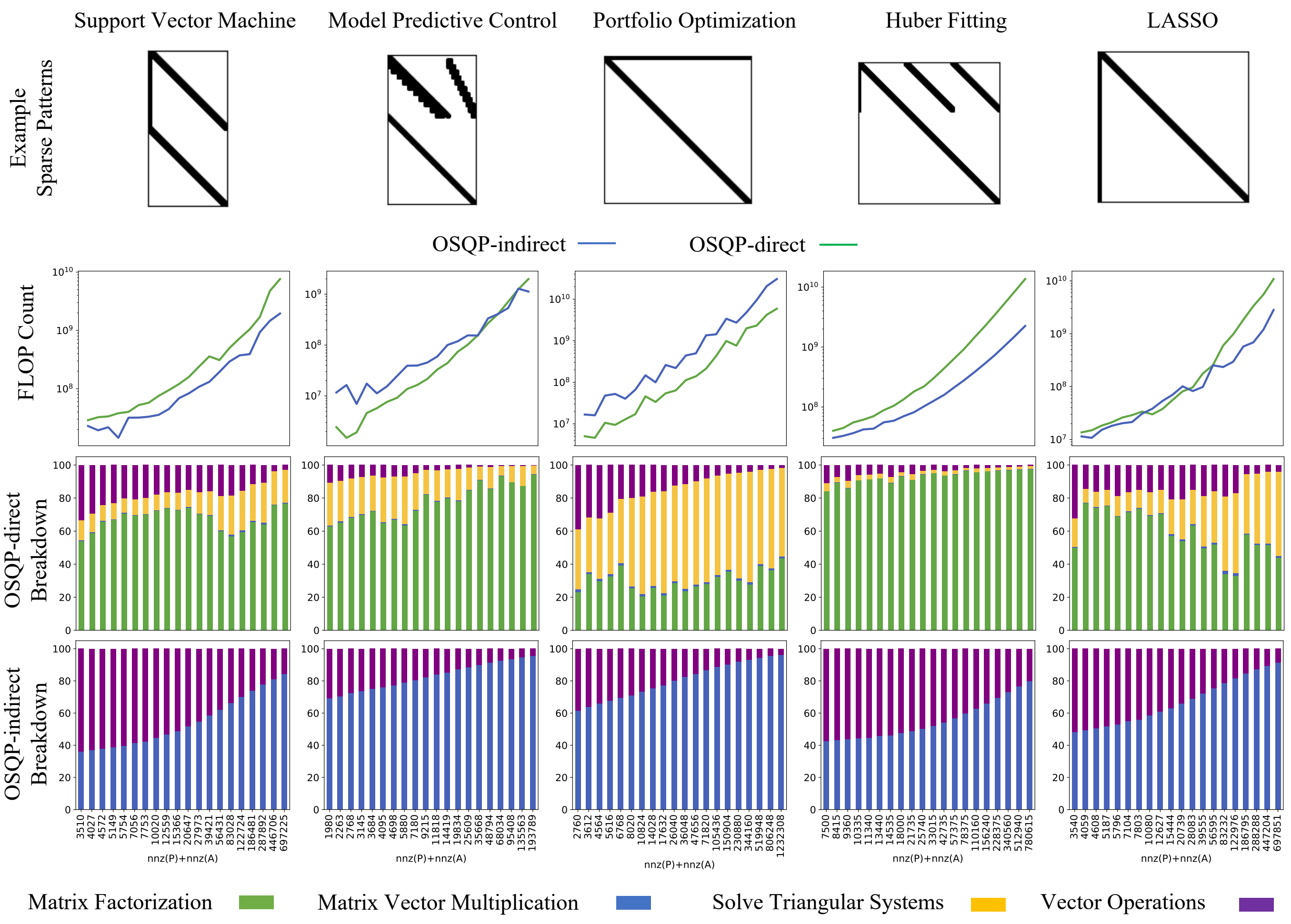


Fig. 3. The top row shows the sparsity patterns that will be shared across problem instances in different domains. The second row shows the total FLOPS count of the two solver algorithm variants. The third and fourth row shows the FLOPS breakdown of different operations.

network efficiently processes C items per clock cycle, with a delay of $\log C$.

To accommodate all potential network configurations, $2C \log C$ bits are required, with each node necessitating 2 bits of control information to select from four operations: pass cross, pass direct or pass the sum of cross and direct. By restricting the network to execute only a select few common computation patterns, the number of bits needed for high-level control can be significantly reduced. The control bits for these patterns can be stored on-chip and replayed to the network upon receiving high-level network instructions.

Figure 6 illustrates the four primitive operations executed by the network. To determine the control signal needed for various operations, the network inputs and outputs are initially encoded using the binary representation of their locations. With 8 inputs and outputs, each location requires 3 bits for encoding. For instance, if we want x_0 at input location 0 to be routed to output location 3, as shown in Figure 6(c), we perform the *xor* operation between the input and output encodings. The resulting control signal, 011, is required for each hop node. The LSB 1 indicates that the first network stage should be set in pass-cross mode, while the MSB 0 signifies that the final network stage should be set in pass-direct mode. The control signals for the remaining input-output pairs are calculated similarly using the xor operation.

For the MAC operation depicted in Figure 6a, all network inputs share the same output location. First, we compute the control signal using the xor operation and set the node collides to pass-sum work mode. For instance, input x_0 and x_1 collides at the second node (marked in orange) at H1, so we designate it as pass-sum. The network topology ensures that if two inputs have the same destination, they will follow the same path after the first collision node. The remaining pass-sum nodes are marked based on calculated collisions as well. It is important to note that MAC can write back to any memory bank and perform output alignment during accumulation. Similarly, column elimination in Figure 6b can select x_0 from any bank and distribute it. The example in the figure demonstrates the case where $C = 8$. The network can be scaled up to $C = 16, 32, 64, ...$ to increase parallelism.

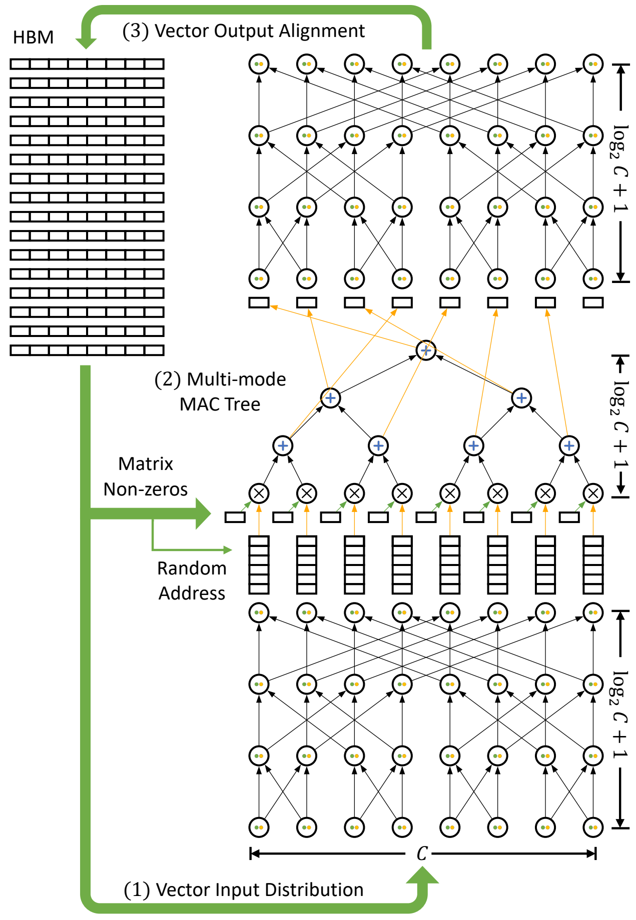


Fig. 4. The baseline architecture for the MAC primitive. The green arrow represents the contiguous data flow from HBM. The orange data path involves random data access.

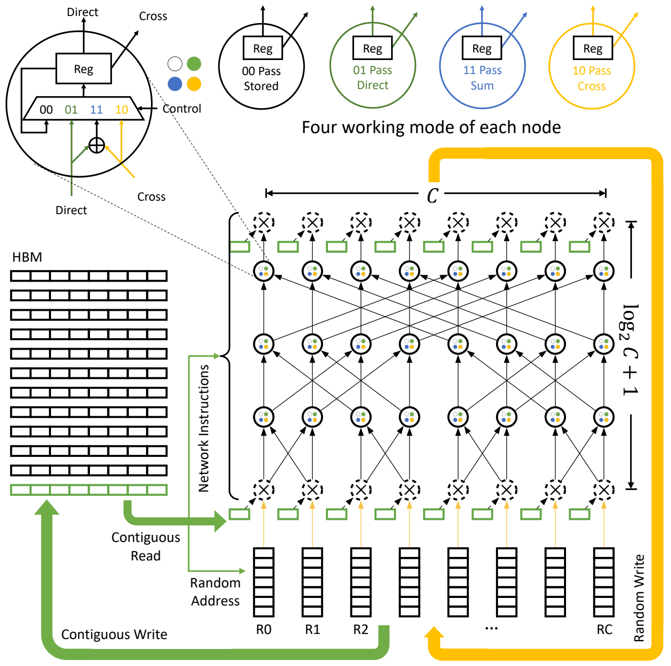


Fig. 5. Computational network supporting all the primitives required by the core operation set. Each multi-mode node, as depicted in the top left, features two inputs and two outputs, labeled as 'direct' and 'cross'. Depending on the network instruction, both outputs are broadcasted either with the 'direct' input, the 'cross' input, or the sum of both.

Networks with a bigger C require larger bandwidth provided by more HBM channels. Note that each node in the network has two fan-in and two fan-out connections, regardless of the C. This property allows for easy scaling of the network width, given enough HBM channels.

D. Programming Model

We developed a two-level instruction set integrated with a compiler framework to map various solver algorithms onto our proposed architecture. Table I lists the top-level instructions required to execute the solver algorithm entirely on this architecture. The compiler accepts both the solver algorithm representation and the sparsity patterns of problem matrices as input.

To facilitate the migration of existing solver C code to our architecture, we adopted a custom C format to represent algorithm computation flow. Listing 1 provides an example of the source program. For simplicity, some function arguments related to the lengths and addresses of vectors and matrices are omitted. The source file compiles into top-level instructions that operate on entire matrices and vectors. These instructions execute sequentially, performing matrix and vector operations one by one. The top-level program is shared across different problem domains and doesn't need to be recompiled.

```
void main(){
  /* defining network instructions to be scheduled */
  net_schedule permutate, inverse_permutate;
  net_schedule L_solve, Lt_solve, D_solve;
  net_schedule A_multiply;
  /* defining vectors  */
  vectorf xtilde_view, ztilde_view, prev_x, data_q;
  /* defining scalars */
  float prim_res, dual_res, sigma;
  /* vector operations */
  xtilde_view = sigma * prev_x - data_q;
  /* matrix multiplication */
  load_vec(xtilde_view, ..., ...);
  net_compute(A_multiply, ...);
  write_vec(ztilde_view, ..., ...);
  /* solving the triangular system*/
  load_vec(xtilde_view, ..., ...);
  load_vec(ztilde_view, ..., ...);
  net_compute(permutate, ...);
  net_compute(L_solve, ...);
  net_compute(D_solve, ...);
  net_compute(Lt_solve, ...);
  net_compute(inverse_permutate, ....);
  write_vec(xtilde_view, ..., ...);
  write_vec(ztilde_view, ..., ...);
}
```

Listing 1. An example source program.

The sparsity patterns of the problem matrices are provided as input to the compiler for low-level network instruction scheduling. The keyword `net_schedule` in the source program specifies the sparsity pattern to be exploited. Top-level

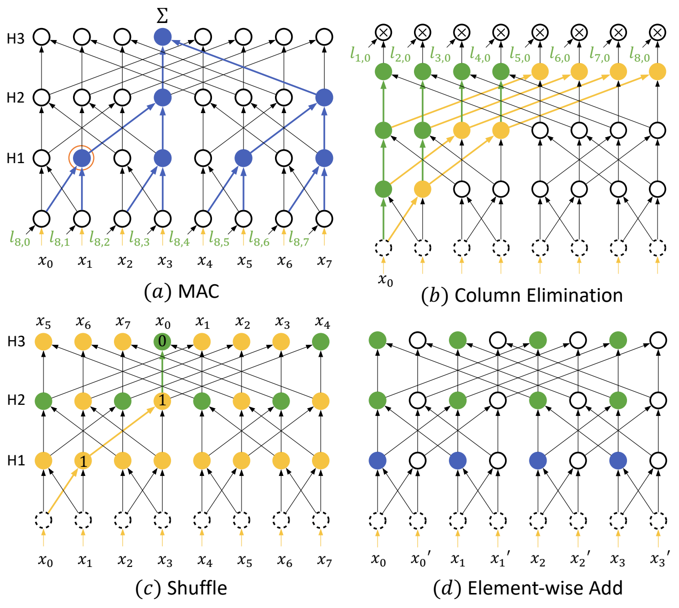


Fig. 6. The network instructions for primitive operations are computed offline. The overall network program will be fed into the network together with the data items.

TABLE I
INSTRUCTION SET

Instruction	Inputs	Computation
`norm_inf`	$v1$	$\|v1\|_\infty$
`cond_set`	$s0, s1, v0, v1$	set vector values
`ew_reci`	$v0$	element-wise reciprocal
`ew_prod`	$v0$	element-wise product
`axpby`	$s0, s1, v0, v1$	$s0v0 + s1v1$
`select_min`	$v0, v1$	select max
`select_max`	$v0, v1$	select min
`net_compute`	$n0, a0$	network compute
`load_vec`	$v0, s0, a0$	vector HBM to register files
`write_vec`	$v0, s0, a0$	vector register files to HBM

instructions involving our computation network are decomposed into numerous low-level network instructions. These network instructions leverage sparsity patterns in various optimization problems, coordinating the working modes of all nodes within the unified computation network. *Multiple-issue* and *re-ordering* of network instructions are essential for achieving rapid end-to-end solver run times. We will explore these concepts in greater detail in the following section.

IV. Network Instruction Scheduling

There is a significant amount of instruction-level parallelism available in the core operation set of network processing. For instance, the calculation of distinct rows in matrix-vector multiplication (line `14` in Listing 1) can be executed independently. Additionally, the permutation of the right-hand side vector elements across various register files can be parallelized both before and after solving triangular systems (line `19` and line `23` in Listing 1).

To increase the instruction throughput, we need to tackle the structural and data hazard when issuing multiple network instructions together. The scheduling problem in our proposed architecture resembles classic multi-issue pipeline processors [19], [36]. In this work, we mainly explore the static scheduling style and leave dynamic scheduling to future work.

A. Structural and Data Hazards

We demonstrate two types of pipeline hazards in our proposed architecture, as shown in Figure 7. We use the notation `I(t, n)(Ri,...)⇢(Rj,...)` to represent the `n`th network instruction issued at clock cycle `t`, which reads from register files `(Ri,...)` and writes to register files `(Rj,...)`. Structural hazards occur due to conflicting access to hardware resources, such as register files or nodes within the network. For instance, both `I(N+1,1)` and `I(N+1,2)` need to read from `R4`, but `R4` can only supply one item per clock cycle. This conflict can be resolved through data prefetching. We can search previous instruction slots and insert a data prefetching instruction, such as `I(1,3)`, to move the operand to `R3` beforehand. Then, `I(N+1,1)` can be rewritten to access `R3` instead, avoiding conflict with `I(N+1,2)`.

A similar structural hazard can occur when writing to the same output register, such as `I(N+2,1)` and `I(N+2,2)`. A comparable solution involves rerouting the entire `I(N+2,1)` to write to a temporary location like `R3` instead, and appending a data prefetching instruction at a later clock cycle when `R4` is unoccupied to write the temporary result back to the original destination. It is worth noting that our architecture can execute these data prefetching instructions without incurring additional hardware costs.

Data hazards occur due to computational dependencies among network instructions. A network instruction's input might not be readily available due to pipeline delays. For example, `I(N+2,2)` needs to read the result of `I(N+1,2)`, but it won't be available until `N+4`. To resolve this, empty instructions can be inserted before the instruction until the result is available. The empty instruction slots can be filled by other independent instructions or data prefetching instructions.

B. The Scheduling Strategy for OSQP-indirect

The OSQP-indirect variant is mostly composed of matrix-vector multiplication instructions as shown in Figure 3. Note that we need to multiply A^T and A, so the multiplication of A is performed with the MAC primitive instruction and A^T is performed with column elimination instruction. Sparsity patterns like the diagonal constraints as shown in Figure 2 are compiled to short network instructions whose widths are less than the hardware network width. Structural hazards are the major obstacles to issuing multiple short network instructions together.

We use bin packing to model the instruction scheduling problem. The hardware resource request of each network instruction is encoded as a binary vector of length $C(\log_2 C+1)$. This vector marks the usage of every node in the network. We

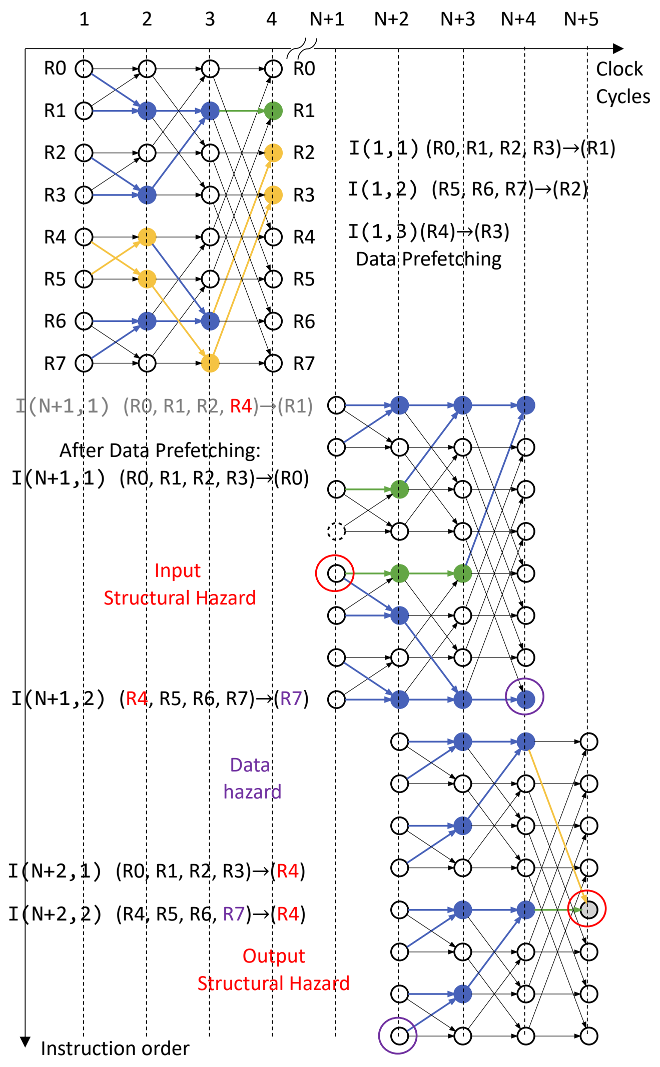


Fig. 7. Different types of pipeline hazards.

use the First-Fit algorithm to solve the above bin packing problem and get a structural hazard-free solution. We start from an initial schedule without multiple issuing. Then go through this initial order from the beginning. For each instruction, we find the first instruction slot that doesn't have conflict with its hardware occupancy vector. We repeat this process until the last instruction. Figure 8 illustrates this spatial and temporal interleave scheduling in detail.

C. The Scheduling Strategy for OSQP-direct

The matrix factorization and triangular solver instructions face more data hazards. The associated data dependency graph has orders of magnitude more edges compared to the matrix multiplication case, as illustrated in the right part of Figure 8 . The research community [2], [11]–[13], [17], [25] has established systematic methods to analyze computation dependencies in sparse matrix factorization. A data structure known as the elimination tree [24], which is the spanning tree of the data dependency graph, can greatly simplify this dependency analysis. We use the elimination tree to get an initial order that satisfies the computation dependencies. Then apply the same first-fit scheduling method.

V. Evaluation

A. System Setup

We create two prototypes of our proposed architecture on the Xilinx Alveo U50 FPGA, with network widths of $C = 16$ and $C = 32$. This FPGA board has 872K LUTs, 1743K Registers, 5,952 DSPs, and 8GB HBM. Figure 9 displays the hardware resource usage percentages. The deep blue areas represent device maps of unified computation networks. Due to the misalignment between our proposed network's butterfly topology and the grid DSP layout on the FPGA, the network's floating-point adders and multipliers are mapped to LUTs and Registers. This results in designs approaching the FPGA's maximum clock frequency of 300 MHz. In the future, the increased layout flexibility provided by ASICs will enable the implementation of larger network widths.

The prototypes are integrated into a system with QP solvers running on other baseline architectures. For comparison, we select the CPU and GPU models with similar process nodes. The specifications can be found in Table II.

We conduct a comparison with the most recent heterogeneous solver, RSQP [42]. This solver enhances PCG performance on the FPGA side, while the remainder of the solver operates on the CPU side.

We also compared our prototypes with the state-of-the-art open-sourced solvers OSQP [38] supporting both CPU and GPU backends. OSQP can use either well-optimized linear algebra libraries to accelerate the sparse matrix operations, such as Intel's Math Kernel Library (MKL) [41] on the CPU and NVIDIA's cuSparse library [28] on the GPU, or standard built-in sparse matrix operations on the CPU (hand-coded matrix operations and the QDLDL factorization).

At present, OSQP does not offer a direct solver with its GPU backend. As demonstrated in [39], [40], GPU-based direct solvers tend to perform poorly on optimization workloads, yielding only a minor speedup or even a slowdown. For the same reason, popular commercial QP solvers such as Gurobi [21] and MOSEK [4] do not currently support GPU backends.

Our prototype's software stack is constructed using an open-source C compiler [6]. We have augmented the compiler's backend with our proposed instruction set, as presented in Table I, enabling the generation of executable files for both OSQP-direct and OSQP-indirect algorithms on our prototypes. To address matrix sparsity patterns, we have further enhanced the compiler by scheduling network instructions based on the strategies outlined in Section IV. Moreover, we have developed a performance profiling tool that examines the Intermediate Representation (IR) produced by our compiler and integrates runtime statistics, such as code block loop count, to generate a comprehensive computational analysis of the two algorithm variations, as depicted in Figure 3.

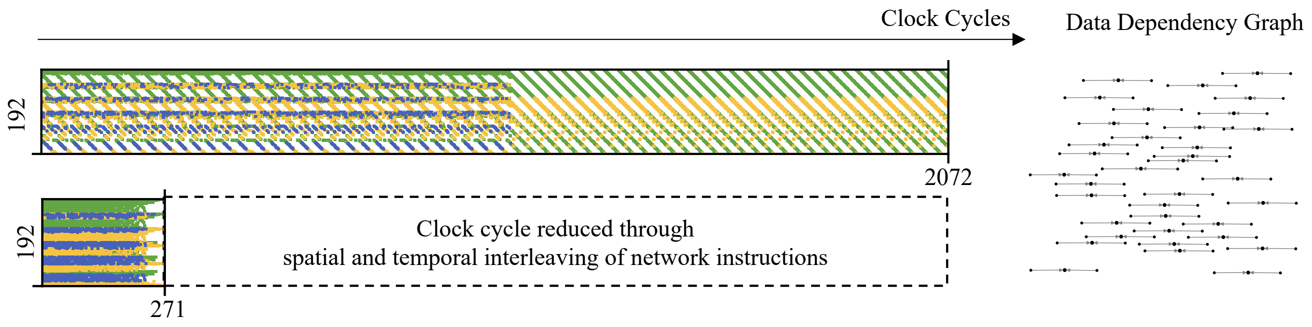
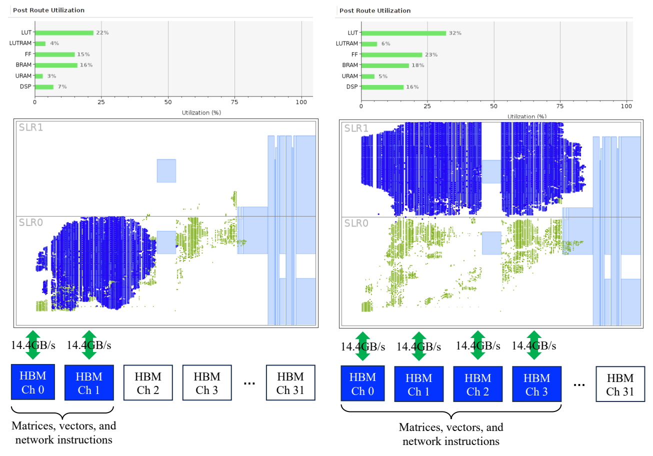

Clock Cycles
Data Dependency Graph
192
2072
192
Clock cycle reduced through
spatial and temporal interleaving of network instructions
271

Fig. 8. The top left section illustrates an example network program before reordering. This network program performs matrix multiplication for matrix A in the SVM domain. Each column represents a network instruction that specifies the operating modes of all 192 nodes within the network, which has a width of $C = 32$. On the right, each network instruction is depicted as a node in the data dependency graph, taking pipeline hazard constraints into account. Network instructions without dependencies are omitted from the graph. The bottom left section displays the network program after reordering, which significantly reduces total execution clock cycles from 2072 to 271.

TABLE II
ARCHITECTURE SPECIFICATIONS

Architecture	This work	This work	RSQP [42]	CPU	GPU
Model	$C = 16$	$C = 32$	Multiple	i7-10700KF	RTX 3070
Process	16 nm	16 nm	16 nm	14 nm	8 nm
Clock	300 MHz	236 MHz	94 to 236 MHz	3.80 GHz	1.75 GHz
FLOPS	33G	60G	8.9 to 15.1 G	500G	20T
Bandwidth	28.8GB/s	57.6GB/s	28.8 to 115.2 GB/s	45.8GB/s	448GB/s
TDP	75W	75W	75W	125W	220W
Library	Ours	Ours	Custom	MKL, QDLDL	cuSparse



(a) Prototype network width $C = 16$ (b) Prototype network width $C = 32$

Fig. 9. Prototypes Resource Usage and Device Map

The input data for the problem and the compiled executable are sent to the prototypes via PCIe. Once the solver algorithm is complete, the prototypes return the results to the CPU. The prototypes have no data communication with the CPU while running the solver algorithm. RSQP [42] incurs additional communication costs during every loop step in Algorithm 1, as it transfers the linear system solution vector between the CPU and FPGA, while the GPU backend [35] sends scalar values from the GPU to the CPU multiple times per loop step to execute the algorithm's control flow operations. To assess the advantages of the proposed architecture, we examine the end-to-end solver speed, power efficiency, and timing reliability.

B. End-to-end Solver Run time

The compiler requires a few seconds to perform network instruction scheduling based on the sparsity pattern. Notably, the sparsity pattern for each application remains constant across different problem *instances*. For example, in the portfolio optimization example discussed in Section II, millions of QPs with the same sparsity pattern must be solved each trading day [29]. Consequently, the time spent compiling the sparsity pattern can be amortized over these numerous instances. RSQP requires the reconfiguration of FPGA for different sparsity patterns and will even incur longer compilation overhead.

Figure 10 shows the end-to-end solver algorithm run time comparison of different platforms when solving 100 real-world QP problems from different application domains. The solver stops when both the primal and dual residual are below a preset threshold. Compared to the OSQP-indirect running on CPU backends, our prototype ($C = 32$) achieves a geometric mean of 30.5$\times$ end-to-end speedup. When compared to the GPU backend, the prototype attains a geometric mean of 4.3$\times$ end-to-end speedup. In the case of OSQP-direct, our

TABLE III
IMPROVEMENT OF THE PROPOSED SOLVER OVER OSQP ON CPU AND GPU

Variant	Baseline	End-to-end runtime speedup	Device Energy Efficiency	System Energy Efficiency	Jitter reduction
OSQP-indirect	GPU (cuSparse)	4.3×	21.7×	9.5×	33.4×
	CPU (MKL)	30.5×	127.0×	37.3×	16.5×
	RSQP	9.5×	N/A	N/A	N/A
OSQP-direct	CPU (QDLDL)	2.7×	11.2×	3.3×	13.8×

prototype achieves a geometric mean of 2.7× end-to-end speedup compared with the CPU with QDLDL. We also include the solver run times for the best architectures generated by RSQP across different application domains. Our solution outperforms RSQP in all domains, achieving a geometric mean end-to-end speedup of 9.5×. This significant improvement is primarily due to the elimination of communication costs between the CPU and the FPGA at each ADMM iteration.

We employ the peak FLOP utilization of each platform as a normalized indicator of architectural efficiency. Our proposed architecture attains a higher overall utilization compared to the CPU and GPU, resulting in a faster solver performance despite having a lower peak FLOP capability.

C. Power Efficiency

We utilize the number of problems solved per second per watt as a normalized indicator for the power efficiency across the different platforms. The power consumption of the CPU, GPU, and our prototypes are measured using the command line tools `powertop`, `nvidia-smi`, and `xbutil`, respectively. Each problem is executed 20 times recording the average power trace to then calculcate the throughput per watt at the end. The FPGA consumes 12W in idle state and approximately 18W under full load. The GPU consumes about 30W in idle state and around 65W under full load. The CPU consumes roughly 22W in idle state and about 49W under full load.

When compared to the OSQP-indirect running on CPU backends, our prototype ($C = 32$) achieves a geometric mean of 127.0× greater energy efficiency. In comparison to the GPU backend, the prototype attains a geometric mean of 21.7× higher energy efficiency. For OSQP-direct, our prototype achieves a geometric mean of 11.2× higher energy efficiency compared to the CPU. The enhanced energy efficiency of our proposed architecture proves advantageous for both edge applications, such as Model Predictive Control, and cloud applications, including portfolio optimization and machine learning algorithms like SVM, LASSO, and Huber fitting.

The above evaluation only considers the power of the standalone device. Since the FPGA and GPU backend still need the CPU to transfer the data at the beginning, we also measured the total system power, including the CPU's idle power. Our work still shows energy efficiency gains, as shown in Table III.

D. Deterministic Timing

In control engineering, deterministic timing is a highly desirable property. To assess this feature, we calculated the standard deviation of solving time normalized by the absolute solver run time for problems in the MPC benchmark across each architecture. The result is shown in Figure 11. When compared to the OSQP-indirect running on CPU backends, our prototype achieves a geometric mean of 16.5× less runtime jitter. In comparison to the GPU backend, the prototype attains a geometric mean of 33.4× less runtime jitter. The deterministic timing, along with the faster solving time of our solution, can facilitate more reliable and smoother control in various applications.

VI. RELATED WORKS

We now briefly discuss the other ADMM QP solvers and sparse linear system solvers developed for other workloads in the literature. A systolic array-based accelerator specifically designed for Cholesky and LU factorization was proposed in [18]. Their approach primarily targets large-scale problems characterized by numerous super nodes in the elimination tree, which is not a typical scenario in the optimization problems studied here. On the other hand, [37] introduced an FPGA-based PCG accelerator, but their solution does not incorporate any specialized optimization techniques for handling matrix sparsity patterns.

There are many works focusing on accelerating specific operators related to the QP solver, like sparse matrix vector multiplication [5], [27], [30]–[32], [34], [44]–[46]. In terms of complete QP solver acceleration, the closet works we can identify are cuOSQP [35] for GPUs and RSQP [42] for FPGAs. RSQP introduced an automated design process that generates an FPGA-assisted QP solver tailored to specific applications, with function units and data paths that match the matrix sparsity structure. The solver architecture remains static while solving each problem but must be reconfigured when switching between problem sparsity structures to get the best performance. The reconfiguration cost is offset by reusing the static architecture for numerous parameterized problems that share the same sparsity structure but have different non-zero values. However, their solution needs to go back and forth between the CPU and FPGA in every ADMM iteration and suffers performance issues on small scale problems. Besides, their architecture only supports OSQP-indirect and lacks the

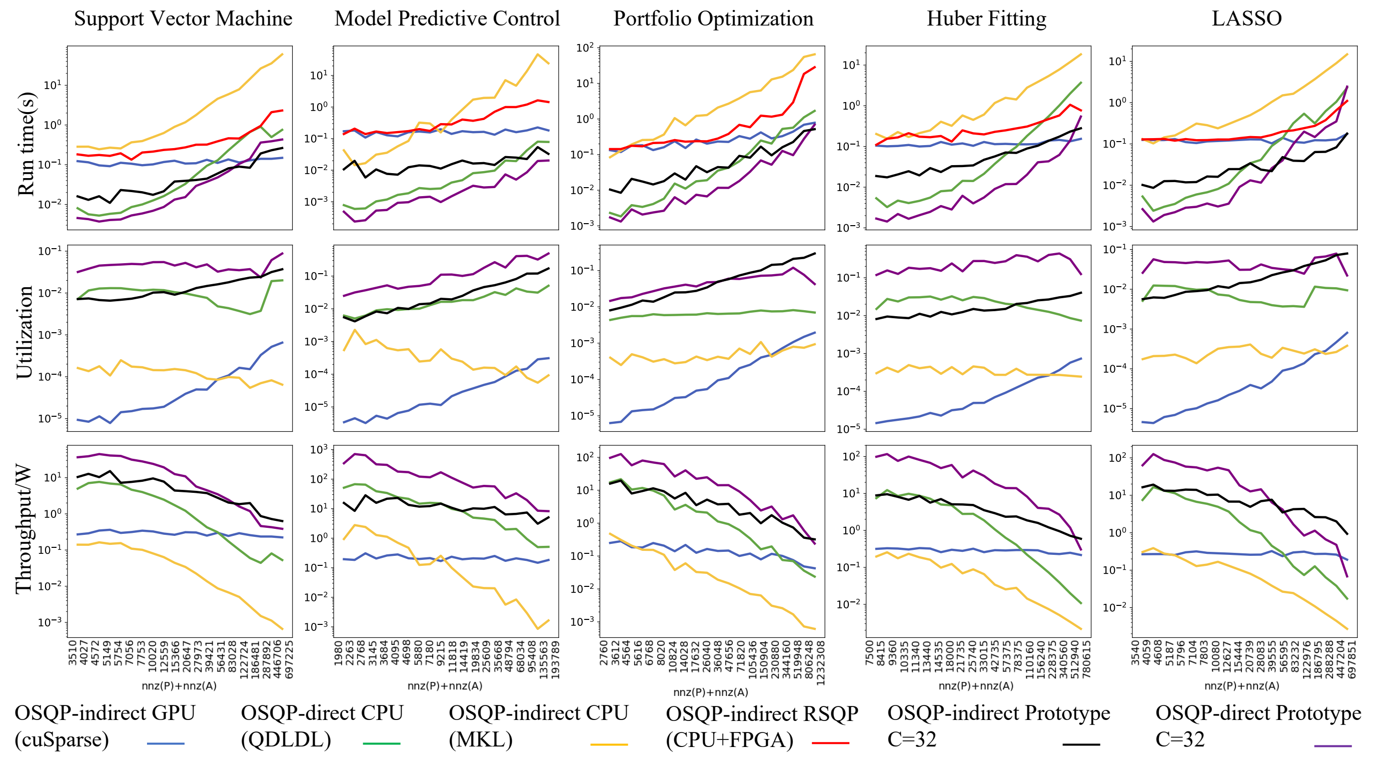


Fig. 10. The top row presents a comparison of solver run times. OSQP currently does not support OSQP-direct on GPUs. Additionally, RSQP generates different architectures for different application domains, so we have excluded it from the utilization and energy efficiency figures, which evaluate a single architecture across different problems.

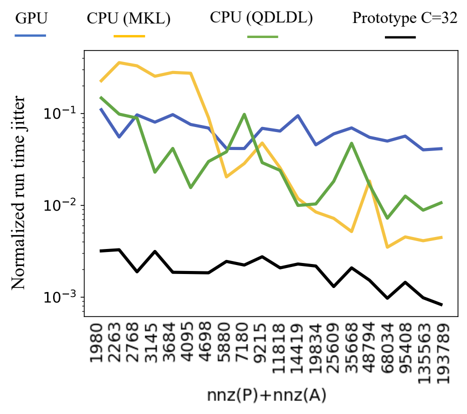


Fig. 11. Run time jitter comparison. The reduction of jitter is due to our cycle-accurate control of the program execution.

support of OSQP-direct. Table IV provides a summary comparison of the latest ADMM-based QP solvers on different platforms. Our work is the first fully FPGA-based generic QP solver.

VII. Conclusion

In conclusion, this study has introduced a novel architecture specifically designed for addressing sparse convex quadratic programs. The proposed solution effectively adapts to varying sparsity patterns by employing customized network instruction scheduling at compile time.

As we look ahead, our research will explore advanced techniques such as super pipelining and dynamic multiple-instruction-issue and reordering. Additionally, we aim to identify and map further application algorithms that share the same core sparse operations. Ultimately, our goal is to develop and validate an ASIC version of the prototypes, further demonstrating the advantages of our proposed architecture.

TABLE IV
Generic QP solvers

	Platform	Architecture Optimization
OSQP	CPU	General Purpose
cuOSQP	CPU+GPU	Sparse Matrix Multiplication
RSQP	CPU+FPGA	Sparse Matrix Multiplication
This work	full-FPGA or ASIC	Sparse Matrix Multiplication and Factorization

References

[1] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter, “Differentiable convex optimization layers,” in *Advances in Neural Information Processing Systems*, 2019.

[2] P. R. Amestoy, T. A. Davis, and I. S. Duff, “Algorithm 837: Amd, an approximate minimum degree ordering algorithm,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 30, no. 3, pp. 381–388, 2004.

[3] B. Amos and J. Z. Kolter, “Optnet: Differentiable optimization as a layer in neural networks,” in *International Conference on Machine Learning*. PMLR, 2017, pp. 136–145.

[4] M. ApS, *The MOSEK optimization toolbox for MATLAB manual. Version 9.0.*, 2019. [Online]. Available: http://docs.mosek.com/9.0/toolbox/index.html

[5] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, “Alrescha: A lightweight reconfigurable sparse-computation accelerator,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 249–260.

[6] E. Bendersky, “Complete c99 parser in pure python,” 2024. [Online]. Available: https://github.com/eliben/pycparser

[7] P. T. Boggs and J. W. Tolle, “Sequential Quadratic Programming,” *Acta Numerica*, vol. 4, pp. 1–52, 1996.

[8] S. Boyd, E. Busseti, S. Diamond, R. N. Kahn, K. Koh, P. Nystrup, and J. Speth, “Multi-Period Trading via Convex Optimization,” *Foundations and Trends® in Optimization*, vol. 3, no. 1, pp. 1–76, Aug. 2017.

[9] M.-A. Boéchat, J. Liu, H. Peyrl, A. Zanarini, and T. Besselmann, “An architecture for solving quadratic programs with the fast gradient method on a field programmable gate array,” in *21st Mediterranean Conference on Control and Automation*, 2013, pp. 1557–1562.

[10] P. Chen, W. Guan, and P. Lu, “Esvio: Event-based stereo visual inertial odometry,” *IEEE Robotics and Automation Letters*, vol. 8, no. 6, pp. 3661–3668, 2023.

[11] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, “Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 35, no. 3, pp. 1–14, 2008.

[12] T. A. Davis, *Direct methods for sparse linear systems*. SIAM, 2006.

[13] T. A. Davis, S. Rajamanickam, and W. M. Sid-Lakhdar, “A survey of direct methods for sparse linear systems,” *Acta Numerica*, vol. 25, pp. 383–566, 2016.

[14] W. Deng, P. Polak, A. Safikhani, and R. Shah, “A Unified Framework for Fast Large-Scale Portfolio Optimization,” *Data Science in Science*, vol. 3, no. 1, p. 2295539, Dec. 2024.

[15] S. Diamond and S. Boyd, “Cvxpy: A python-embedded modeling language for convex optimization,” *The Journal of Machine Learning Research*, vol. 17, no. 1, pp. 2909–2913, 2016.

[16] Z. Dong, W. Mau, Y. Feng, Z. T. Pennington, L. Chen, Y. Zaki, K. Rajan, T. Shuman, D. Aharoni, and D. J. Cai, “Minian, an open-source miniscope analysis pipeline,” *Elife*, vol. 11, p. e70661, 2022.

[17] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct methods for sparse matrices*. Oxford University Press, 2017.

[18] A. Feldmann and D. Sanchez, “Spatula: A hardware accelerator for sparse matrix factorization,” in *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*, 2023, pp. 91–104.

[19] J. A. Fisher, “Very long instruction word architectures and the eli-512,” in *Proceedings of the 10th annual international symposium on Computer architecture*, 1983, pp. 140–150.

[20] P. Grigoraş, P. Burovskiy, W. Luk, and S. Sherwin, “Optimising sparse matrix vector multiplication for large scale FEM problems on FPGA,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–9.

[21] Gurobi Optimization, LLC, “Gurobi Optimizer Reference Manual,” 2023. [Online]. Available: https://www.gurobi.com

[22] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, “Embedded predictive control on an FPGA using the fast gradient method,” in *2013 European Control Conference (ECC)*, 2013, pp. 3614–3620.

[23] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, “An FPGA implementation of a sparse quadratic programming solver for constrained predictive control,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’11. New York, NY, USA: Association for Computing Machinery, 2011, p. 209–218.

[24] J. W. Liu, “The role of elimination trees in sparse factorization,” *SIAM journal on matrix analysis and applications*, vol. 11, no. 1, pp. 134–172, 1990.

[25] J. W. Liu, E. G. Ng, and B. W. Peyton, “On finding supernodes for sparse matrix computations,” *SIAM Journal on Matrix Analysis and Applications*, vol. 14, no. 1, pp. 242–252, 1993.

[26] I. McInerney, G. A. Constantinides, and E. C. Kerrigan, “A survey of the implementation of linear model predictive control on FPGAs,” *IFAC-PapersOnLine*, vol. 51, no. 20, pp. 381–387, 2018.

[27] F. Muñoz-Martínez, R. Garg, M. Pellauer, J. L. Abellán, M. E. Acacio, and T. Krishna, “Flexagon: A multi-dataflow sparse-sparse matrix multiplication accelerator for efficient dnn processing,” in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 252–265.

[28] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, “Cusparse library,” in *GPU Technology Conference*, 2010.

[29] P. Nystrup, S. Boyd, E. Lindström, and H. Madsen, “Multi-period portfolio selection with drawdown control,” *Annals of Operations Research*, vol. 282, no. 1-2, pp. 245–271, 2019.

[30] S. Pal, A. Amarnath, S. Feng, M. O’Boyle, R. Dreslinski, and C. Dubach, “Sparseadapt: Runtime control for sparse linear algebra on a reconfigurable accelerator,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 2021, pp. 1005–1021.

[31] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski, “Outerspace: An outer product based sparse matrix multiplication accelerator,” in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 724–736.

[32] G. Rao, J. Chen, J. Yik, and X. Qian, “Sparsecore: stream isa and processor specialization for sparse computation,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 186–199.

[33] J.-M. Rodriguez-Bernuz, I. McInerney, A. Junyent-Ferré, and E. C. Kerrigan, “Design of a Linear Time-Varying Model Predictive Control Energy Regulator for Grid-Tied VSCs,” *IEEE Transactions on Energy Conversion*, vol. 36, no. 2, pp. 1425–1434, 2021.

[34] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, “Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 347–358.

[35] M. Schubiger, G. Banjac, and J. Lygeros, “GPU acceleration of ADMM for large-scale quadratic programming,” *Journal of Parallel and Distributed Computing*, vol. 144, pp. 55–67, 2020.

[36] J. E. Smith and G. S. Sohi, “The microarchitecture of superscalar processors,” *Proceedings of the IEEE*, vol. 83, no. 12, pp. 1609–1624, 1995.

[37] L. Song, L. Guo, S. Basalama, Y. Chi, R. F. Lucas, and J. Cong, “Callipepla: Stream centric instruction set and mixed precision for accelerating conjugate gradient solver,” in *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2023, pp. 247–258.

[38] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, “OSQP: an operator splitting solver for quadratic programs,” *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.

[39] K. Świrydowicz, E. Darve, W. Jones, J. Maack, S. Regev, M. A. Saunders, S. J. Thomas, and S. Peleš, “Linear solvers for power grid optimization problems: a review of gpu-accelerated linear solvers,” *Parallel Computing*, vol. 111, p. 102870, 2022.

[40] K. Świrydowicz, N. Koukpaizan, T. Ribizel, F. Göbel, S. Abhyankar, H. Anzt, and S. Peleš, “Gpu-resident sparse direct linear solvers for alternating current optimal power flow analysis,” *International Journal of Electrical Power & Energy Systems*, vol. 155, p. 109517, 2024.

[41] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, “Intel math kernel library,” in *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 2014, pp. 167–188.

[42] M. Wang, I. McInerney, B. Stellato, S. Boyd, and H. So, “Rsqp: Problem-specific architectural customization for accelerated convex quadratic optimization,” in *2023 ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*, 2023.

[43] S.-X. Wen, Z.-R. Pan, K.-Z. Liu, X. Zhang, and X.-M. Sun, “Practical Offset-Free Model Predictive Control and Its Embedded Application to Aeroengines,” *IEEE Transactions on Automation Science and Engineering*, vol. Early Access, pp. 1–11, 2023.

[44] X. Xie, Z. Liang, P. Gu, A. Basak, L. Deng, L. Liang, X. Hu, and Y. Xie, “Spacea: Sparse matrix vector multiplication on processing-in-memory accelerator,” in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 570–583.

[45] G. Zhang, N. Attaluri, J. S. Emer, and D. Sanchez, “Gamma: Leveraging gustavson’s algorithm to accelerate sparse matrix multiplication,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 687–701.

[46] Z. Zhang, H. Wang, S. Han, and W. J. Dally, “Sparch: Efficient architecture for sparse matrix multiplication,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 261–274.